Tecniche Algoritmiche/3
Programmazione Dinamica

Gianluigi Zavattaro
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna
gianluigi.zavattaro@unibo.it

Slide realizzate a partire da materiale fornito dal Prof. Moreno Marzolla

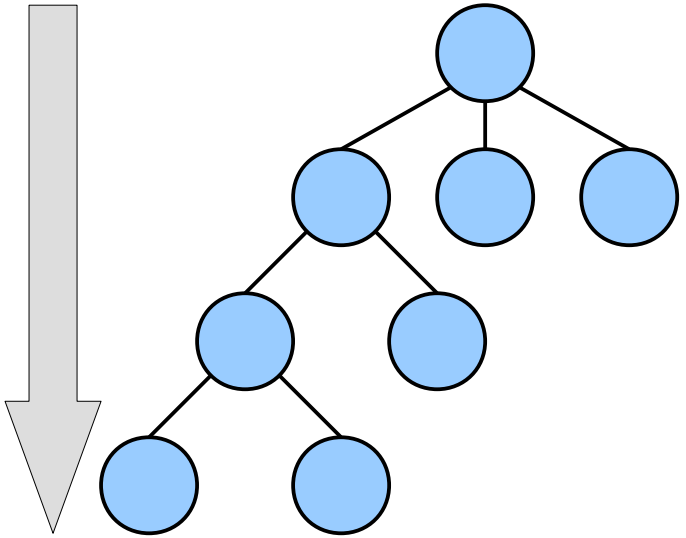
Programmazione dinamica

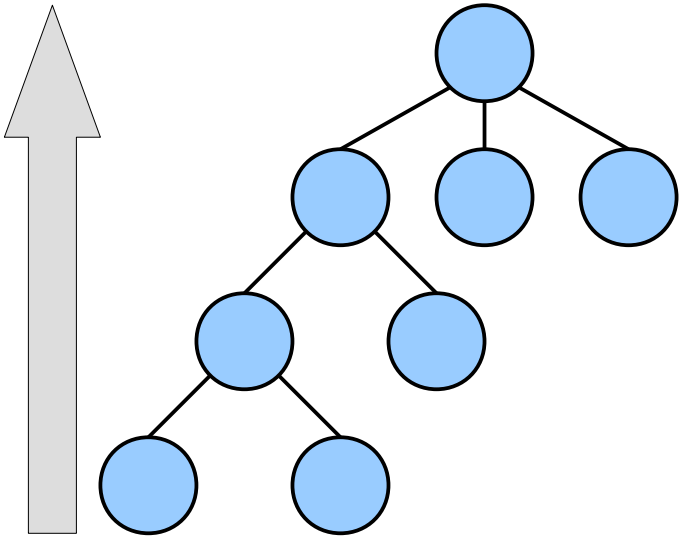
- Definizione
  - Strategia sviluppata negli anni '50 da Richard E. Bellman
  - Ambito: problemi di ottimizzazione
  - Trovare la soluzione ottima secondo un “indice di qualità” assegnato ad ognuna delle soluzioni possibili
- Approccio
  - Risolvere un problema combinando le soluzioni di sotto-problemi
  - Ma ci sono importanti differenze con divide-et-impera

Richard Ernest Bellman (1920—1984), http://en.wikipedia.org/wiki/Richard_Bellman

Programmazione dinamica vs divide-et-impera

- Divide-et-impera
  - Tecnica ricorsiva
  - Approccio top-down
  - Vantaggiosa quando i sottoproblemi sono indipendenti
- Programmazione dinamica
  - Tecnica iterativa
  - Approccio bottom-up
  - Vantaggiosa quando ci sono sottoproblemi ripetuti





Quando applicare la programmazione dinamica?

- Sottostruttura ottima
  - Deve essere possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più "grande"
- Sottoproblemi ripetuti
  - Un sottoproblema compare più volte

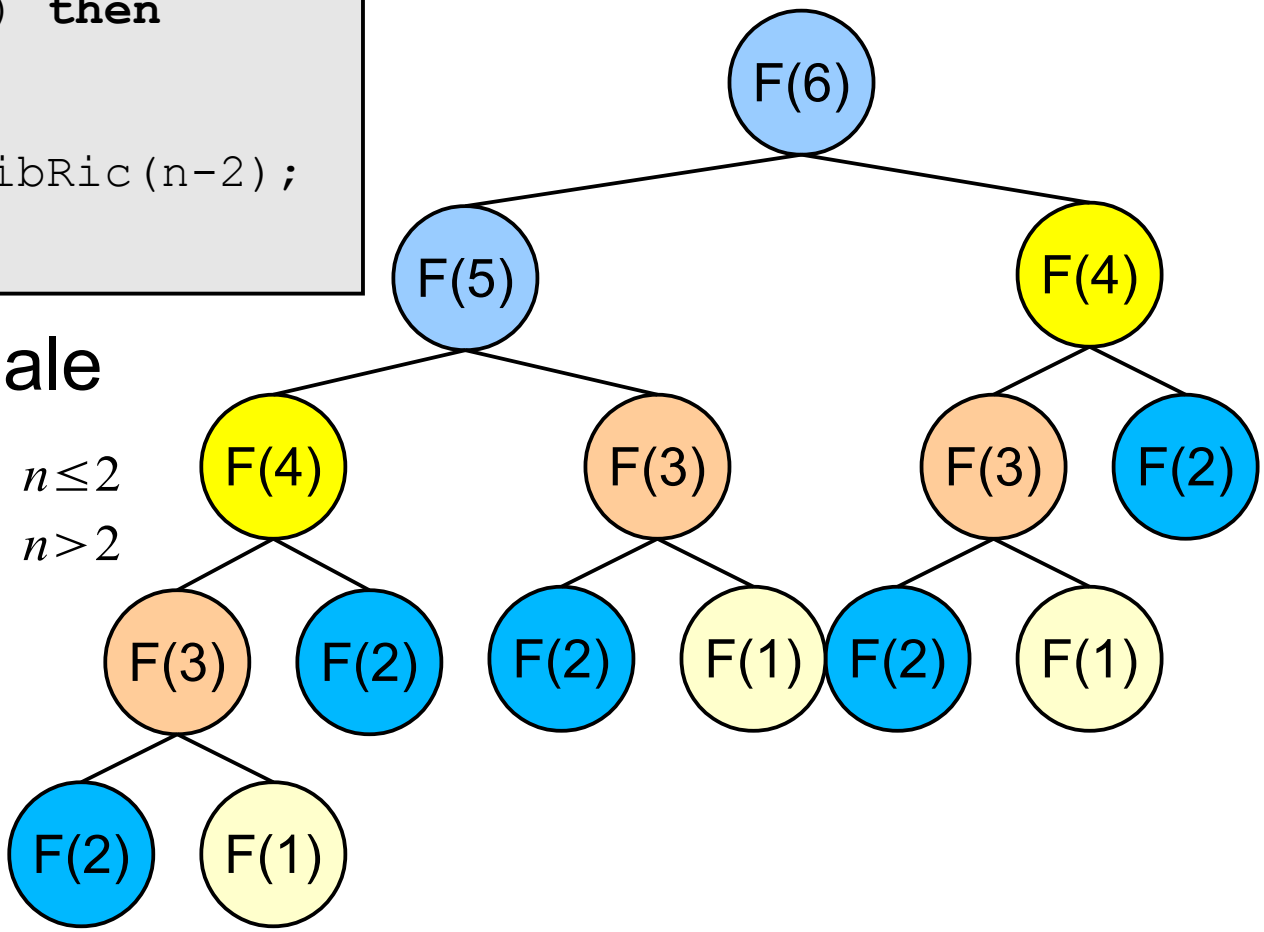
Ricordate: Fibonacci divide-et-impera (top-down)

```
integer FibRic(integer n)
  if ((n = 1) or (n = 2)) then
    return 1;
  else
    return FibRic(n-1)+FibRic(n-2);
  endif
```

- Costo computazionale

$$T(n)=\begin{cases} c_1 & n\leq 2 \\ T(n-1)+T(n-2)+c_2 & n>2 \end{cases}$$

- Soluzione
  - $T(n) = O(2^n)$

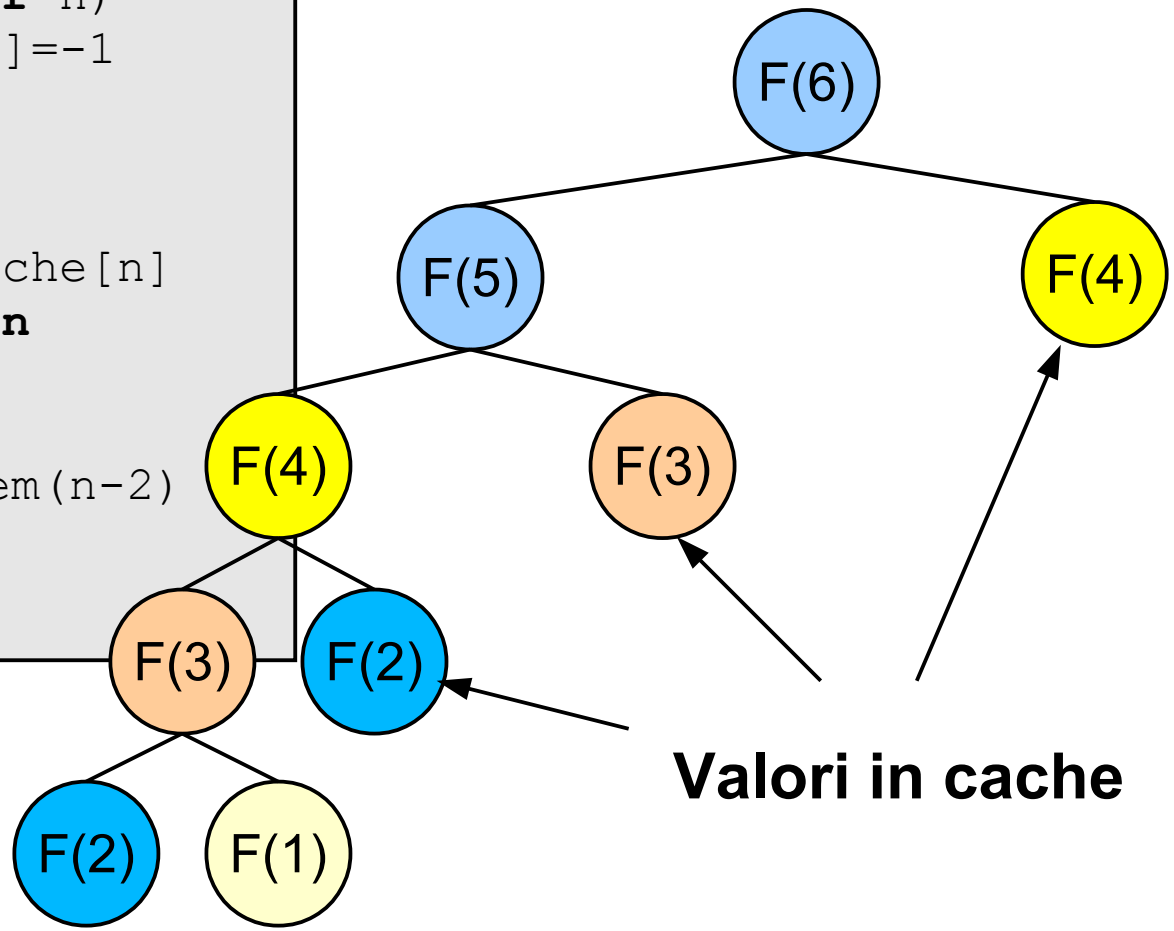


Memoization: cache per evitare di ricalcolare

```
integer cache[n]

integer FibMemoization(integer n)
  for (i=0; i<n; i++) cache[i]=-1
  FibMem(n)

integer FibMem(integer n)
  if (cache[n]!=-1) return cache[n]
  if ((n = 1) or (n = 2)) then
    cache[n]=1
  else
    cache[n]=FibMem(n-1)+FibMem(n-2)
  endif
  return cache[n];
```



- Costo computazionale
  - *2n-3 chiamate ricorsive*
  - *T*(*n*) = *O*(n)

Ricordate: Fibonacci progr. Dinamica (bottom-up)

```
integer FibIter(integer n)
  if (n ≤ 2) then
    return 1;
  else
    integer f[1..n];
    f[1] ← 1;
    f[2] ← 1;
    for integer i ← 3 to n do
      f[i] ← f[i-1] + f[i-2];
    endfor
    return f[n];
  endif
```

- Costo computazionale
  - Tempo: O(*n*)
  - Spazio: O(*n*)

n	1	2	3	4	5	6	7	8
f[]	1	1	2	3	5	8	13	21

Ricordate: Fibonacci progr. Dinamica (bottom-up / ottimizzazione memoria)

```
algoritmo Fib(int n) → int
  if (n<2) then
    return 1;
  else
    f := new array[0..2] of int;
    f[1] := 1; f[2] := 1;
    for i := 2 to n do
      f[0] = f[1];
      f[1] = f[2];
      f[2] := f[1] + f[0];
    endfor
    return f[2];
  endif
```

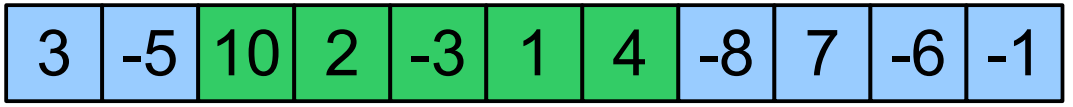
- Costo computazionale
  - Tempo: O*(n)*
  - Spazio: O*(1)*
    - Array di *3* elementi

n	2	3	4	5
f[0]	1	1	2	3
f[1]	1	2	3	5
f[2]	2	3	5	8

Sottovettore di valore massimo

Esempio (già visto) sottovettore di valore massimo

- Consideriamo un vettore $V[1..n]$ di n valori reali arbitrari
- Vogliamo individuare un sottovettore non vuoto di V la somma dei cui elementi sia massima



- Abbiamo già visto (vedi tecniche divide-et-impera) che ci sono $n(n+1)/2$ sottovettori non vuoti di V

Soluzione basata su “forza bruta” Costo $O(n^3)$

```
real SommaMax1( real V[1..n] )
    real smax ← V[1];
    for integer i ← 1 to n do
        for integer j ← i to n do
            real s ← 0;
            for integer k ← i to j do
                s ← s + V[k];
            endfor
            if (s > smax) then
                smax ← s;
            endif
        endfor
    endfor
    return smax;
```

Implementazione più efficiente
Costo $O(n^2)$

```
real SommaMax2( real V[1..n] )
    real smax ← V[1];
    for integer i ← 1 to n do
        real s ← 0;
        for integer j ← i to n do
            s ← s + V[j];
            if (s > smax) then
                smax ← s;
            endif
        endfor
    endfor
    return smax;
```

Imp. basata su divide-et-impera
Costo O(n log n)

```
real SommaMaxDI( real V[1..n], integer i, j )
    if (i>j) return 0
    else if (i==j) return V[i]
    else
        m ← floor((i+j)/2);
        real l ← SommaMaxDI( V, i, m-1 )
        real r ← SommaMaxDI( V, m+1, j )
        real sa ← 0, sb ← 0, s ← 0;
        integer k;
        for (k ← m-1; k>=i; k--) {
            s ← s + V[k];
            if (s > sa) sa ← s;
        }
        s ← 0;
        for (k ← m+1; k<=j; k++) {
            s ← s + V[k];
            if (s > sb) sb ← s;
        }
        return max(l, r, V[m]+sa+sb);
```

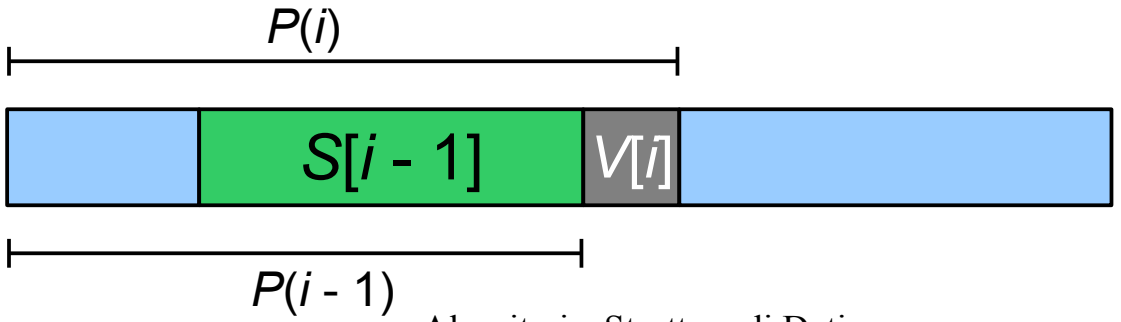
Soluzione basata su programmazione dinamica

- Sia *P*(*i*) il problema che consiste nel determinare il valore massimo della somma degli elementi dei sottovettori non vuoti del vettore *V*[1..*i*] che hanno *V*[*i*] come ultimo elemento
- Sia *S*[*i*] il valore della soluzione di *P*(*i*)
  - *S*[*i*] è la massima somma degli elementi dei sottovettori di *V*[1..*i*] che hanno *V*[*i*] come ultimo elemento
- La soluzione *S* al problema di partenza può essere espressa come

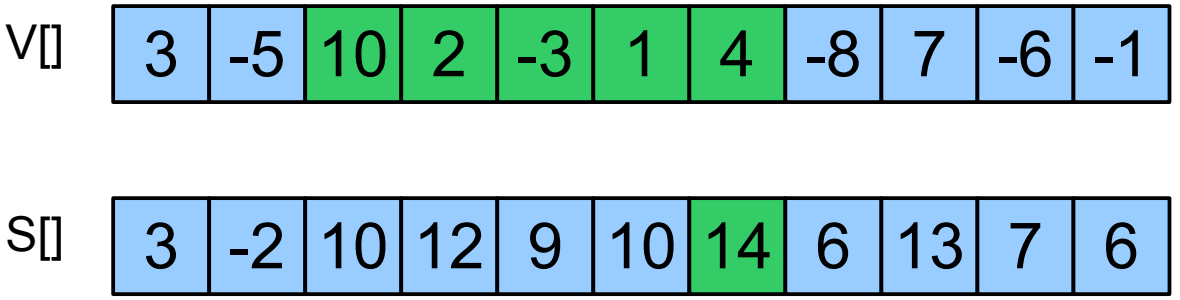
$$S = max_{1 \leq i \leq n} S[i]$$

Soluzione basata su programmazione dinamica

- $P(1)$ ammette una unica soluzione
  - $S[1] = V[1]$
- Consideriamo il generico problema $P(i)$, $i > 1$
  - Supponiamo di avere già risolto il problema $P(i - 1)$, e quindi di conoscere $S[i - 1]$
  - Se $S[i - 1]+V[i] \geq V[i]$ allora $S[i] = S[i - 1]+V[i]$
  - Se $S[i - 1]+V[i] < V[i]$ allora $S[i] = V[i]$



Esempio



$$S[i] = \max \{ V[i], V[\mathrm{i}] + S[i - 1] \}$$

L'algoritmo

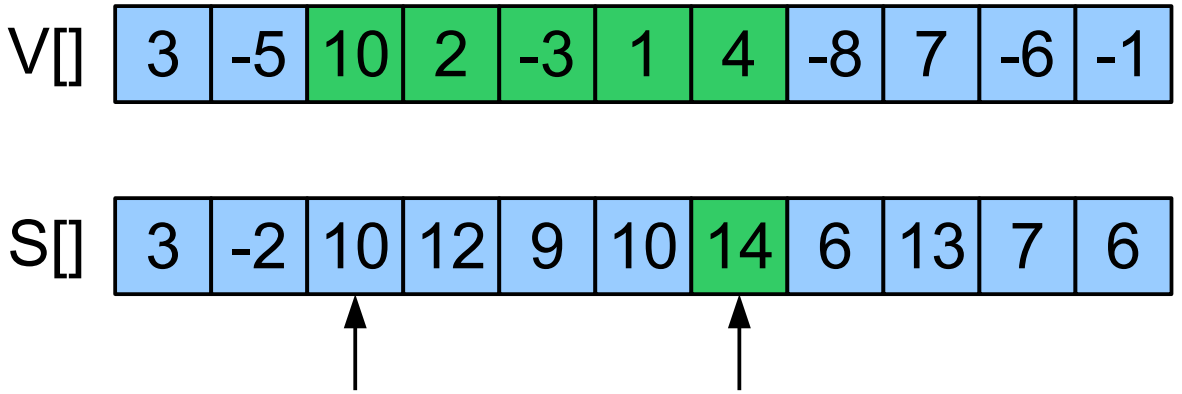
```
real sottovettoreMax(real V[1..n])
    real S[1..n];
    S[1] ← V[1];
    integer imax ← 1;    // indice val. max in S
    for integer i ← 2 to n do
        if ( S[i-1]+V[i] ≥ V[i] ) then
            S[i] ← S[i-1] + V[i];
        else
            S[i] ← V[i];
        endif
        if ( S[i] > S[imax] ) then
            imax ← i;
        endif
    endfor
    return S[imax];
```

Costo?

Come individuare il sottovettore?

- Fino ad ora siamo in grado di calcolare il valore della massima somma tra tutti i sottovettori di $V[1..n]$
- Come facciamo a determinare *quale* sottovettore produce tale somma?
  - L'indice dell'elemento finale del sottovettore l'abbiamo
  - L'indice iniziale lo possiamo ricavare procedendo “a ritroso”
    - Se $S[i] = V[i]$, il sottovettore massimo inizia nella posizione i

Esempio




```
integer indiceInizio(real V[1..n], real S[1..n], integer imax)
    integer i ← imax;
    while ( S[i] ≠ V[i] ) do
        i ← i - 1;
    endwhile
    return i;
```

Problema dello Zaino
(*Knapsack problem*)

Problema dello zaino

- Abbiamo un insieme $X = \{1, 2, ..., n\}$ di n oggetti
- L'oggetto i-esimo ha peso (intero positivo) $p[i]$ e valore $v[i]$
- *Disponiamo di un contenitore in grado di trasporta*re al massimo un peso (intero positivo) P
- Vogliamo determinare un sottoinsieme $Y \subseteq X$ tale che
  - Il peso complessivo degli oggetti in Y sia $\leq P$
  - Il valore complessivo degli oggetti in Y sia il massimo possibile, tra tutti gli insiemi di oggetti che possiamo inserire nel contenitore

Definizione formale

- Vogliamo determinare un sottoinsieme $Y \subseteq X$ di oggetti tale che:

$$\sum_{x \in Y} p[x] \leq P$$

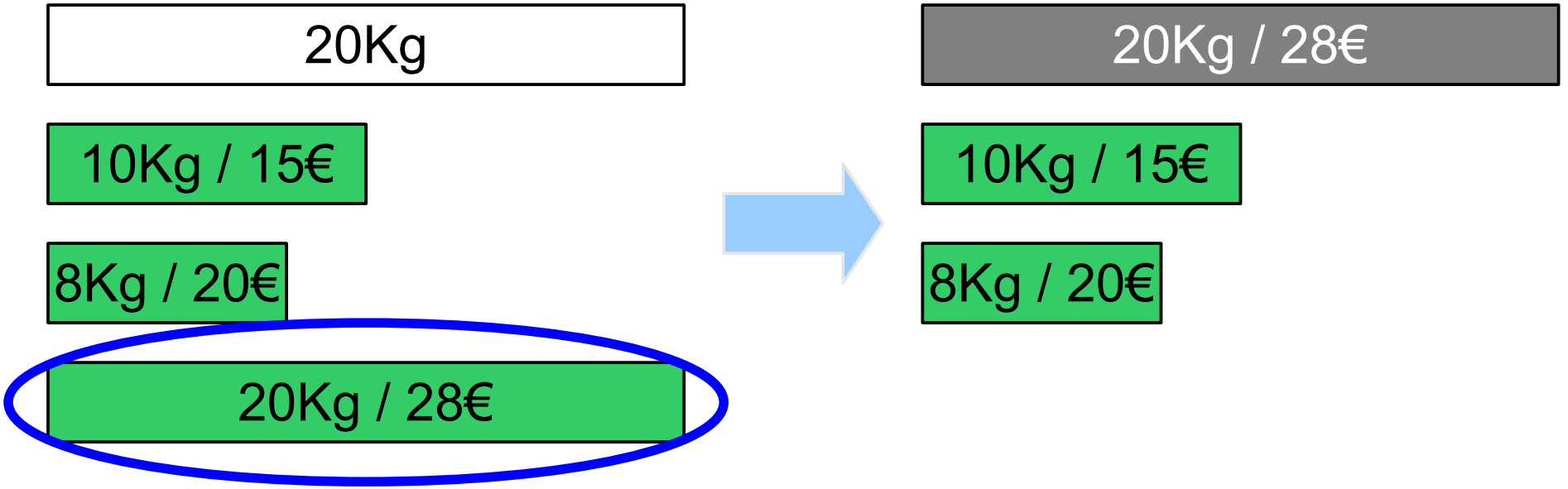
e tale da massimizzare il valore complessivo:

$$\sum_{x \in Y} v[x]$$

Approccio greedy #1
(non produce sempre una soluzione ottima)

- Ad ogni passo, scelgo tra gli oggetti non ancora nello zaino quello di valore massimo, tra tutti quelli che hanno un peso minore o uguale alla capacità residua dello zaino
  - Questo algoritmo non fornisce sempre la soluzione ottima

Esempio

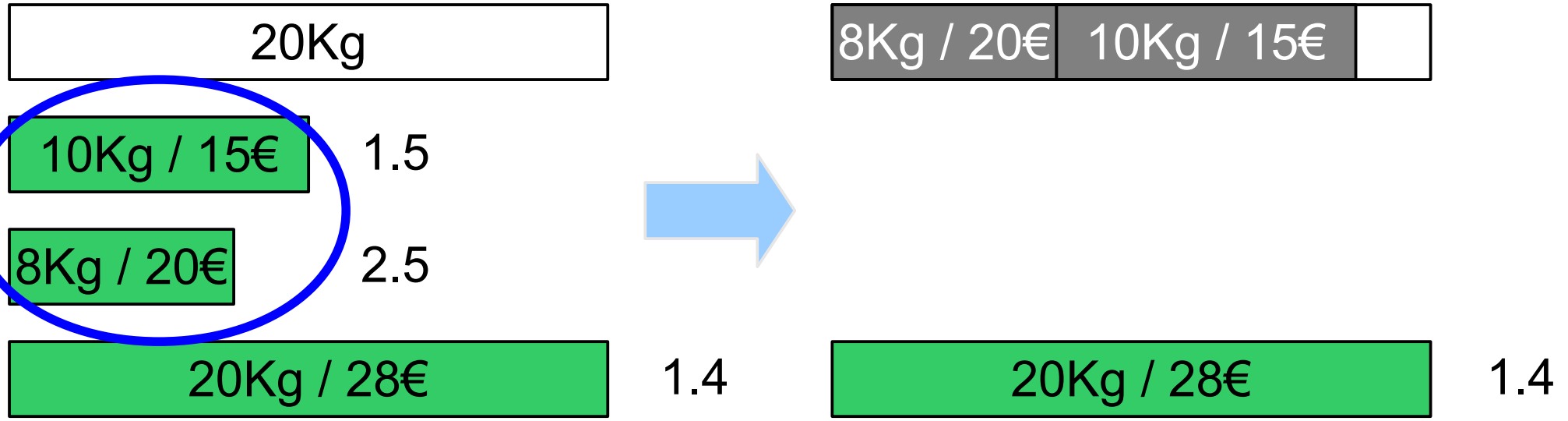


- La soluzione calcolata in questo esempio non è la soluzione ottima!

Approccio greedy #2
(non produce sempre una soluzione ottima)

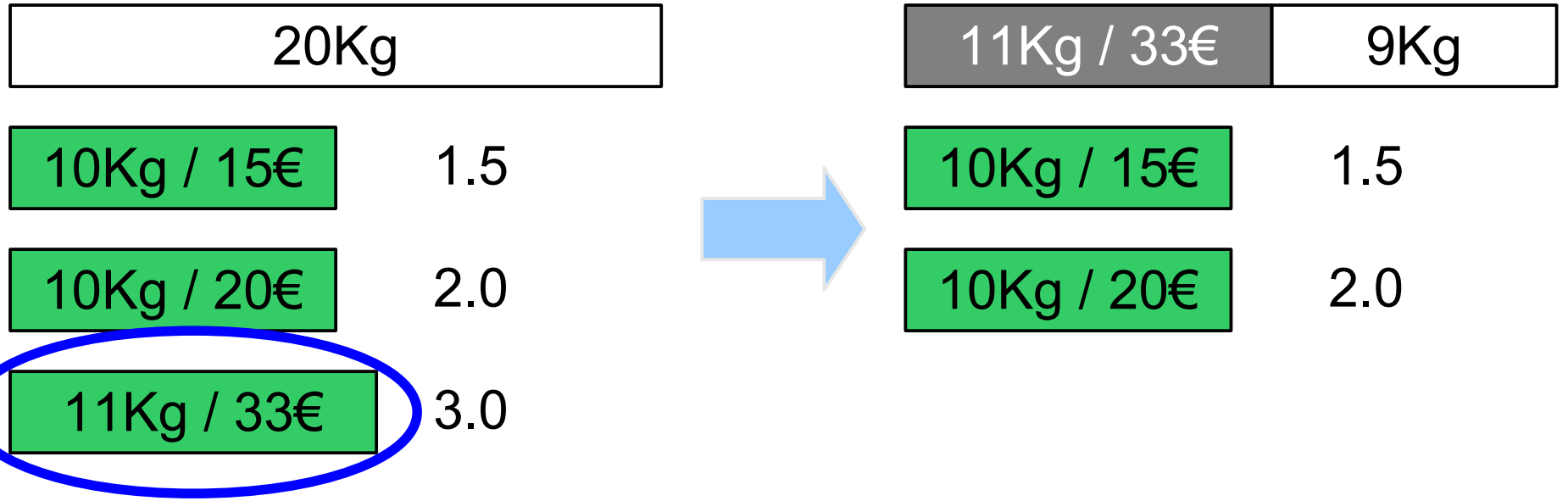
- Ad ogni passo, scelgo tra gli oggetti non ancora nello zaino quello di valore specifico massimo, tra tutti quelli che hanno un peso minore o uguale alla capacità residua dello zaino
  - Il valore specifico è definito come il valore di un oggetto diviso il suo peso
  - Anche questo approccio non fornisce sempre la soluzione ottima

Esempio



- In questo esempio, l'algoritmo greedy #2 calcola la soluzione ottima, ma...

Esempio



- ...in questo altro esempio anche l'algoritmo greedy #2 non produce la soluzione ottima

Soluzione ottima basata sulla Programmazione Dinamica

- NB: l'algoritmo di programmazione dinamica richiede che i pesi siano numeri interi
- Definizione dei sottoproblemi $P(i, j)$
  - *"Riempire uno zaino di capienza j, utilizzando un opportuno sottoinsieme dei primi i oggetti, massimizzando il valore degli oggetti usati"*
- Definizione delle soluzioni $V[i, j]$
  - $V[i, j]$ è il massimo valore ottenibile da un sottoinsieme degli oggetti $\{1, 2, \ldots i\}$ in uno zaino che ha capacità j
  - $i = 1, 2, \ldots n$
  - $j = 0, 1, \ldots P$

Calcolo delle soluzioni: casi base

- Primo caso base: zaino di capienza zero
  - $V[i, 0] = 0$ per ogni $i = 1..n$
    - Se la capacità dello zaino è zero, il massimo valore ottenibile è zero (nessun oggetto)
- Secondo caso base: ho a disposizione solo l'oggetto 1
  - $V[1, j] = v[1]$ se $j \geq p[1]$
    - C'è abbastanza spazio per l'oggetto numero 1
  - $V[1, j] = 0$ se $j < p[1]$
    - Non c'è abbastanza spazio per l'oggetto numero 1

$V[i, j]$ denota il massimo valore ottenibile da un sottoinsieme degli oggetti {1, 2, ..., i} in uno zaino che ha capacità massima j

Calcolo delle soluzioni: caso generale

- Se $j < p[i]$ significa che l'i-esimo oggetto è troppo pesante per essere contenuto nello zaino. Quindi $V[i, j] = V[i - 1, j]$
- Se $j \geq p[i]$ possiamo scegliere se
  - Inserire l'oggetto i-esimo nello zaino. Il valore massimo ottenibile in questo caso è $V[i - 1, j - p[i]] + v[i]$
  - Non inserire l'oggetto i-esimo nello zaino. Il massimo valore ottenibile in questo caso è $V[i - 1, j]$
  - Scegliamo l'alternativa che massimizza il valore: $V[i, j] = \max\{ V[i - 1, j], V[i - 1, j - p[i]] + v[i] \}$

$V[i, j]$ denota il massimo valore ottenibile da un sottoinsieme degli oggetti $\{1, 2, ..., i\}$ in uno zaino che ha capacità massima j

Soluzione ottima basata sulla Programmazione Dinamica

- Riassumendo

$$V[i,j]=\begin{cases} V[i-1,j] & se\ j<p[i] \\ max\{V[i-1,j],V[i-1,j-p[i]]+v[i]\} & se\ j\geq p[i] \end{cases}$$

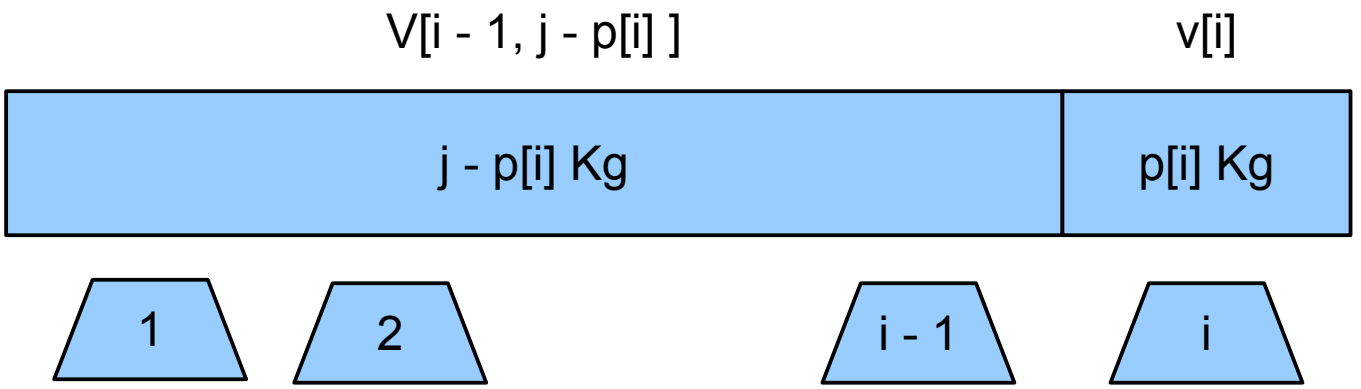


Tabella di programmazione dinamica / matrice V

p =[2, 7, 6, 4]
v =[12.7, 6.4, 1.7, 0.3]

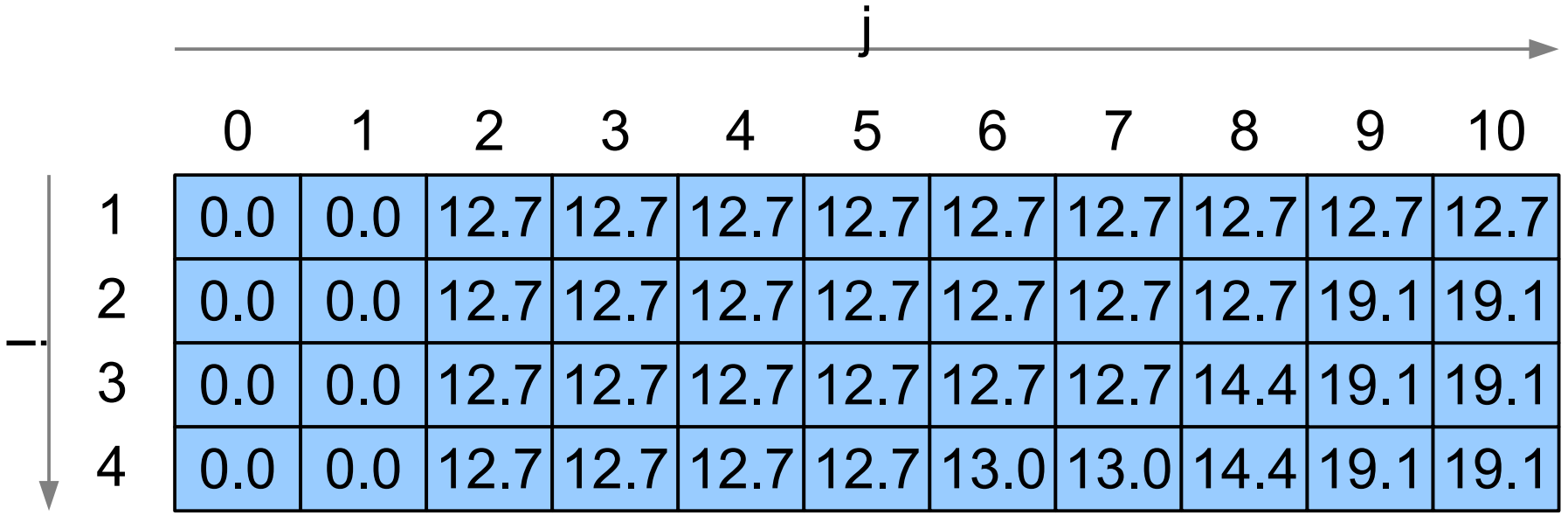
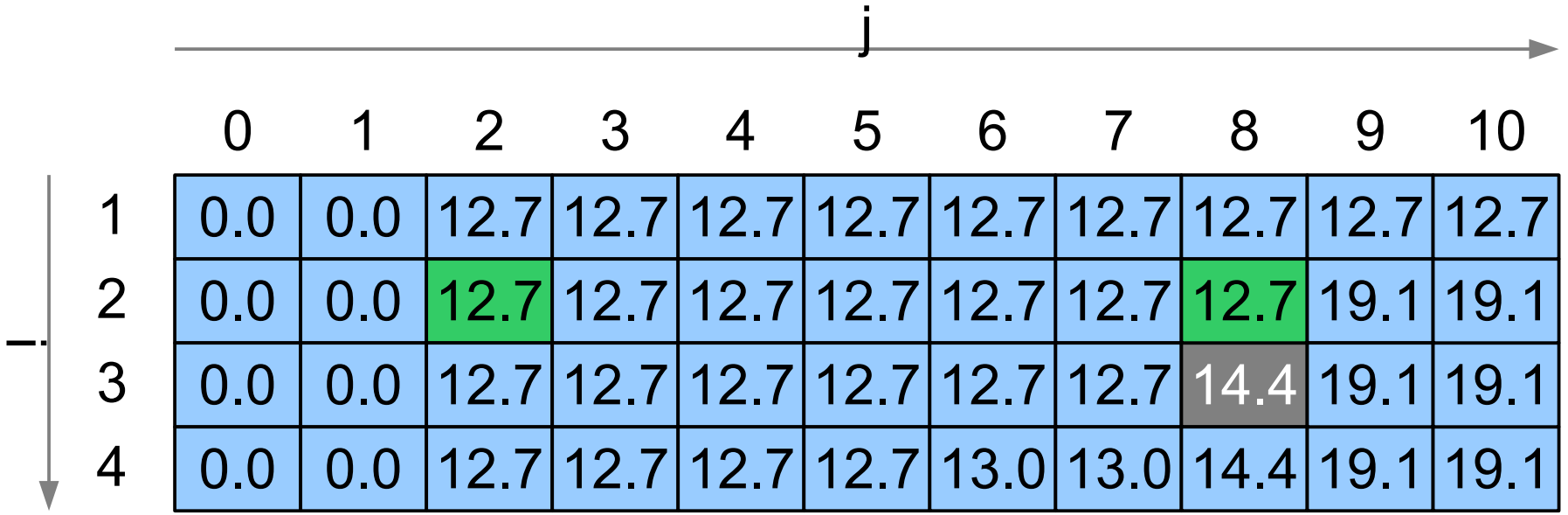


Tabella di programmazione dinamica / matrice V

p =[2, 7, 6, 4]
v =[12.7, 6.4, 1.7, 0.3]



$$V(3,8) = \max\{V(2,8), V(2,8-6)+1.7\}$$
$$= \max\{12.7, 14.4\}$$

Stampare la soluzione

- Il valore della soluzione ottima del problema di partenza è $V[n, P]$
- Come facciamo a sapere quali oggetti fanno parte della soluzione ottima?
  - Usiamo una tabella ausiliaria booleana $K[i, j]$ che ha le stesse dimensioni di $V[i, j]$
  - $K[i, j]$ = *true* se e solo se l'oggetto i-esimo fa parte della soluzione ottima del problema $P(i, j)$ che ha valore $V[i, j]$

Tabella di programmazione dinamica / stampa soluzione ottima

```
integer j ← P;
integer i ← n;
while ( i > 0 ) do
    if ( K[i,j] = true ) then
        stampa "Seleziono oggetto ", i
        j ← j - p[i];
    endif
    i ← i - 1;
endwhile
```

p =[2, 7, 6, 4]
v =[12.7, 6.4, 1.7, 0.3]

	0	1	2	3	4	5	6	7	8	9	10
1	F	F	T	T	T	T	T	T	T	T	T
2	F	F	F	F	F	F	F	F	F	T	T
3	F	F	F	F	F	F	F	F	T	F	F
4	F	F	F	F	F	F	T	T	F	F	F

Seam Carving

Seam Carving

- Algoritmo per ridimensionare immagini in modo “intelligente”
  - Shai Avidan and Ariel Shamir. 2007. *Seam carving for content-aware image resizing*. In ACM SIGGRAPH 2007 (SIGGRAPH '07). ACM, New York, NY, USA, http://doi.acm.org/10.1145/1275808.1276390

https://en.wikipedia.org/wiki/Seam_carving

Scaling

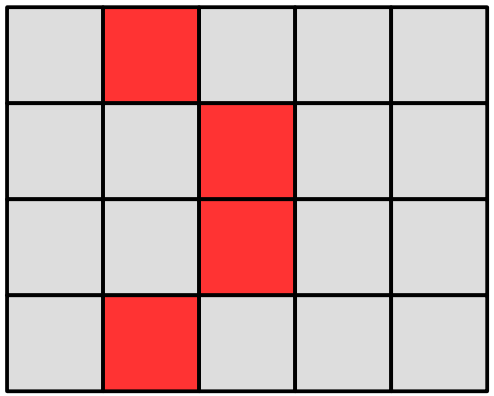
Cropping

Seam Carving

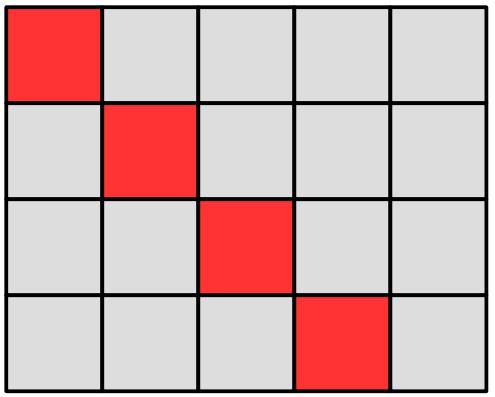
https://en.wikipedia.org/wiki/Seam_carving

Seam Carving

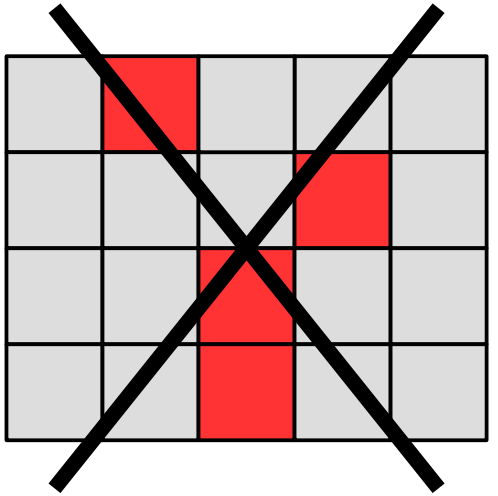
- L'immagine viene ridimensionata togliendo *cuciture*
- Una cucitura (seam) è un cammino composto da pixel adiacenti di “minima importanza”
  - Se l'immagine ha *M* righe per *N* colonne, una cucitura (verticale) è una sequenza di *M* pixel adiacenti, uno per ogni riga



OK



OK



No: pixel non adiacenti

Seam Carving

- Assegnare ad ogni pixel (i, j) un peso $E[i, j] \in [0, 1]$ che denota quanto il pixel è "importante"
  - Es.: quanto un pixel è "diverso" da quelli adiacenti
  - 0 = non importante,
    1 = molto importante
- Determinare una cucitura verticale di peso minimo
- Rimuovere i pixel della cucitura, ottenendo una immagine $M \times (N - 1)$
- Ripetere il procedimento fino ad ottenere la larghezza desiderata.

0.1	0.0	0.2	0.9	0.8
0.9	0.2	0.8	0.4	0.7
0.8	0.8	0.1	0.7	0.8
0.1	0.0	0.6	0.5	0.7

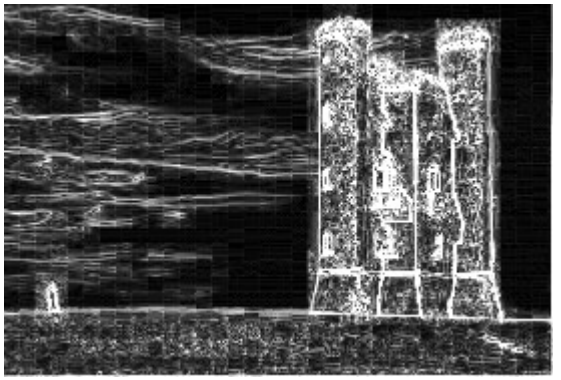
0.1	0.0	0.2	0.9	0.8
0.9	0.2	0.8	0.4	0.7
0.8	0.8	0.1	0.7	0.8
0.1	0.0	0.6	0.5	0.7

0.1	0.2	0.9	0.8
0.9	0.8	0.4	0.7
0.8	0.8	0.7	0.8
0.1	0.6	0.5	0.7

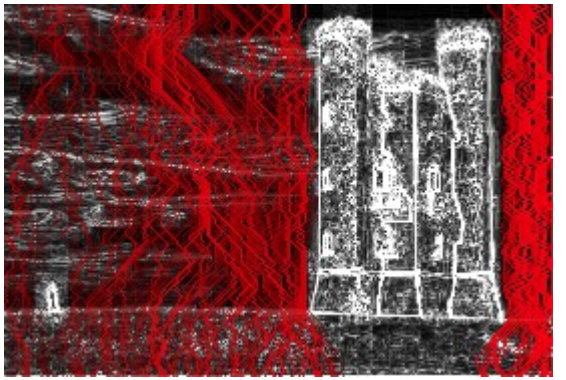
Esempio

Immagine originale

Pesi (nero=0, bianco=1)



Alcune cuciture di peso minimo



https://en.wikipedia.org/wiki/Seam_carving

Determinare le cuciture di peso minimo con la programmazione dinamica

- Definizione dei sottoproblemi $P(i, j)$
  - Determinare una cucitura di peso minimo che termina nel pixel di coordinate (i, j)
- Definizione delle soluzioni $W[i, j]$
  - Minimo peso tra tutte le possibili cuciture che terminano nel pixel di coordinate (i, j)
- Calcolo della soluzione del problema originario
  - La cucitura di peso minimo avrà peso pari al minimo tra $\{ W[M, 1], \ldots W[M, N] \}$

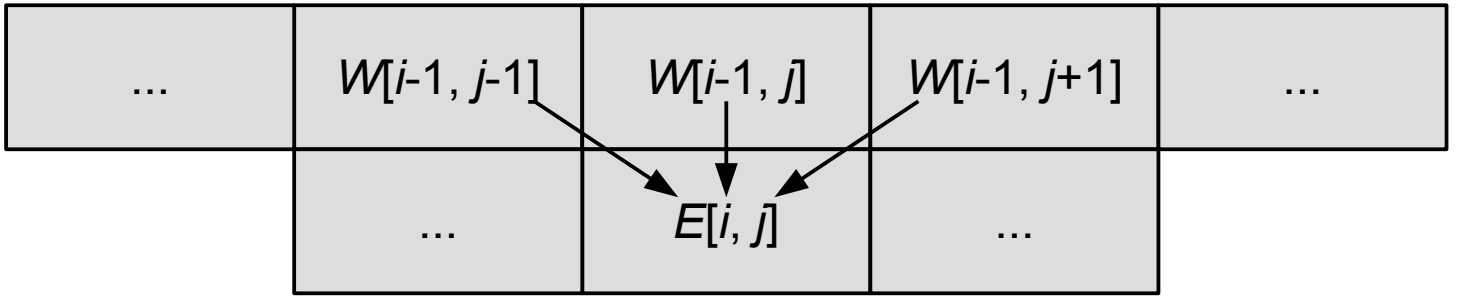
Calcolo di $W[i, j]$

- Casi base ($i = 1$):
  - $W[1, j] = E[1, j]$ per ogni $j = 1, \ldots M$
- Caso generale ($i > 1$):
  - Se $j = 1$

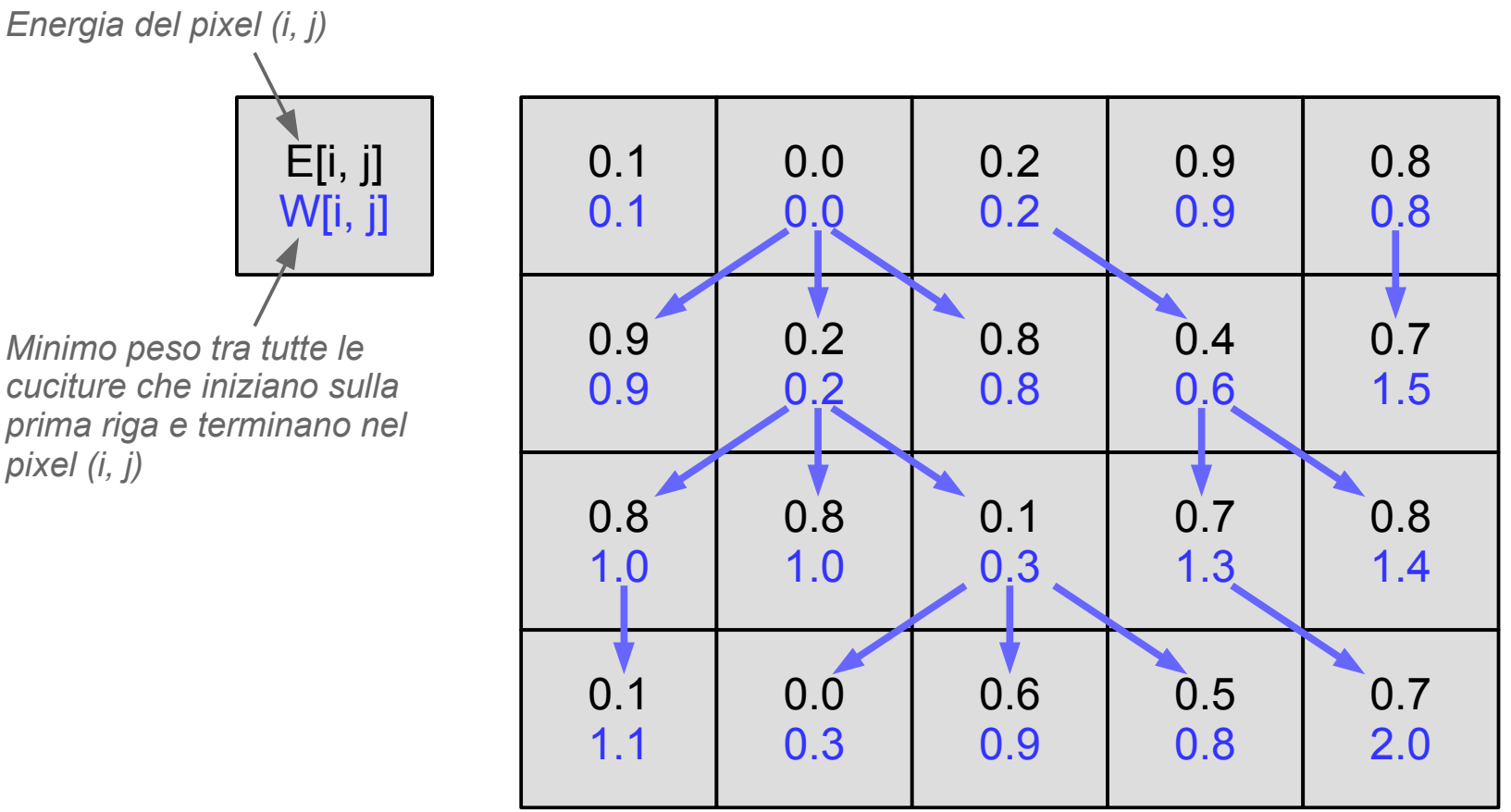
    $W[i, j] = E[i, j] + \min \{ W[i - 1, j], W[i - 1, j + 1] \}$
  - Se $1 < j < N$

    $W[i, j] = E[i, j] + \min \{ W[i - 1, j - 1], W[i - 1, j], W[i - 1, j + 1] \}$
  - Se $j = N$

    $W[i, j] = E[i, j] + \min \{ W[i - 1, j - 1], W[i - 1, j] \}$

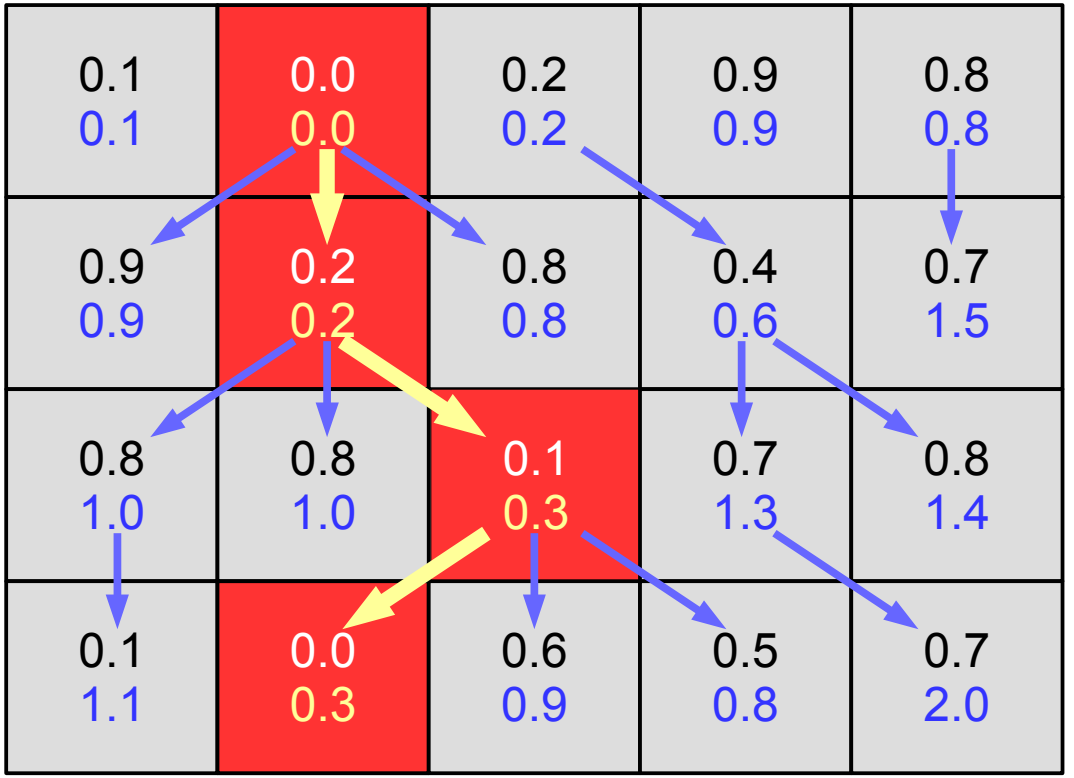


Esempio


Energia del pixel (i, j)
E[i, j]
W[i, j]
Minimo peso tra tutte le cuciture che iniziano sulla prima riga e terminano nel pixel (i, j)
0.1 0.1
0.0 0.0
0.2 0.2
0.9 0.9
0.8 0.8
0.9 0.9
0.2 0.2
0.8 0.8
0.4 0.6
0.7 1.5
0.8 1.0
0.8 1.0
0.1 0.3
0.7 1.3
0.8 1.4
0.1 1.1
0.0 0.3
0.6 0.9
0.5 0.8
0.7 2.0

Esempio

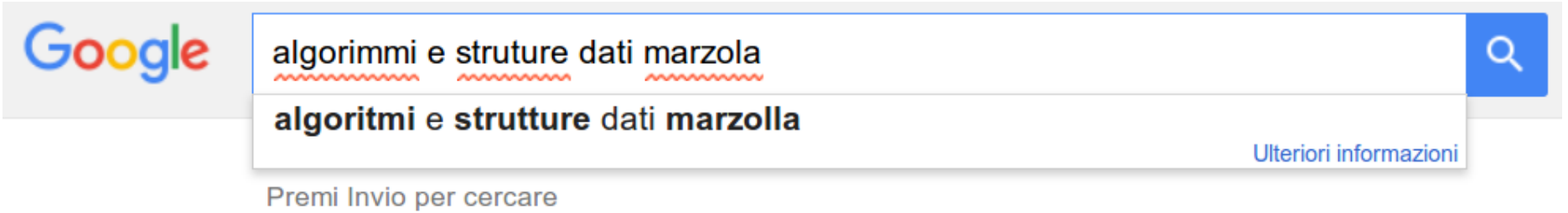
E[i, j]
W[i, j]



Distanza di Levenshtein

Introduzione

- I correttori ortografici sono in grado di suggerire le parole corrette più simili a quello che abbiamo digitato
- Come si fa a decidere quanto “simili” sono due stringhe (=sequenze di caratteri) ?



Distanza di Levenshtein

- Basata sul concetto di "*edit distance*"
  - Numero di operazioni di "editing" che sono necessarie per trasformare una stringa *S* in una nuova stringa *T*
- Trasformazioni ammesse
  - Lasciare immutato il carattere corrente (costo 0)
  - Cancellare un carattere (costo 1)
  - Inserire un carattere (costo 1)
  - Sostituire il carattere corrente con uno diverso (costo 1)
- Dopo ciascuna operazione ci si sposta sul carattere successivo
  - Si inizia dal primo carattere di *S*

Esempio

- Trasformiamo “ALBERO” in “LIBRO”
  - Una possibilità è cancellare tutti i caratteri di “ALBERO” e inserire tutti quelli di “LIBRO”
  - Costo totale: 6 cancellazioni + 5 inserimenti = 11

ALBERO	→	LBERO	cancellazione A
LBERO	→	BERO	cancellazione L
BERO	→	ERO	cancellazione B
ERO	→	RO	cancellazione E
RO	→	O	cancellazione R
O	→	_	cancellazione O
_	→	L_	inserimento L
L_	→	LI_	inserimento I
LI_	→	LIB_	inserimento B
LIB_	→	LIBR_	inserimento R
LIBR_	→	LIBRO_	inserimento O

Esempio

- Possiamo ottenere lo stesso risultato con un costo inferiore
  - 2 cancellazioni + 1 inserimento = 3

ALBERO	→	LBERO	cancello A
LBERO	→	LBERO	lascio immutato
LBERO	→	LIBERO	inserisco I
LIBERO	→	LIBERO	lascio immutato
LIBERO	→	LIBRO	cancello E

Definizione

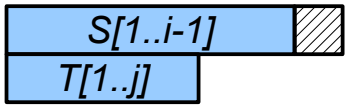
- Due stringhe $S[1..n]$ e $T[1..m]$ di n ed m caratteri, rispettivamente
  - Una o entrambe potrebbero anche essere vuote.
- La distanza di Levenshtein tra $S[1..n]$ e $T[1..m]$ è il costo minimo tra tutte le sequenze di operazioni di editing che trasformano S in T
- Alcune definizioni aggiuntive
  - $S[1..i]$ la stringa composta dai primi i caratteri di S
    - se $i = 0$ è la sottostringa vuota
  - $T[1..j]$ la stringa composta dai primi j caratteri di T
    - se $j = 0$ è la sottostringa vuota

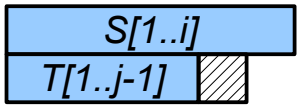
Determinare la distanza di Levenshtein con la programmazione dinamica

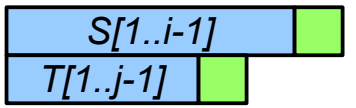
- Definizione dei sottoproblemi $P(i, j)$
  - Determinare il minimo numero di operazioni di editing necessarie per trasformare il prefisso $S[1..i]$ di S nel prefisso $T[1..j]$ di T
- Definizione delle soluzioni $L[i, j]$
  - minimo numero di operazioni di editing necessarie per trasformare il prefisso $S[1..i]$ di S nel prefisso $T[1..j]$ di T
- Calcolo della soluzione del problema originario
  - La distanza di Levenshtein tra $S[1..n]$ e $T[1..m]$ è il valore $L[n, m]$

Calcolo di $L[i, j]$

- Se $i = 0$ oppure $j = 0$
  - Il costo per trasformare una stringa vuota in una non vuota è dato dalla lunghezza della stringa non vuota (occorre inserire i caratteri necessari)
- Se $i > 0$ e $j > 0$, il minimo costo tra:
  - Trasformare $S[1..i - 1]$ in $T[1..j]$, e cancellare l'ultimo carattere $S[i]$ di S
  - Trasformare $S[1..i]$ in $T[1..j - 1]$ e inserire l'ultimo carattere $T[j]$ di T
  - Trasformare $S[1..i - 1]$ in $T[1..j - 1]$ e cambiare $S[i]$ in $T[j]$ se diversi

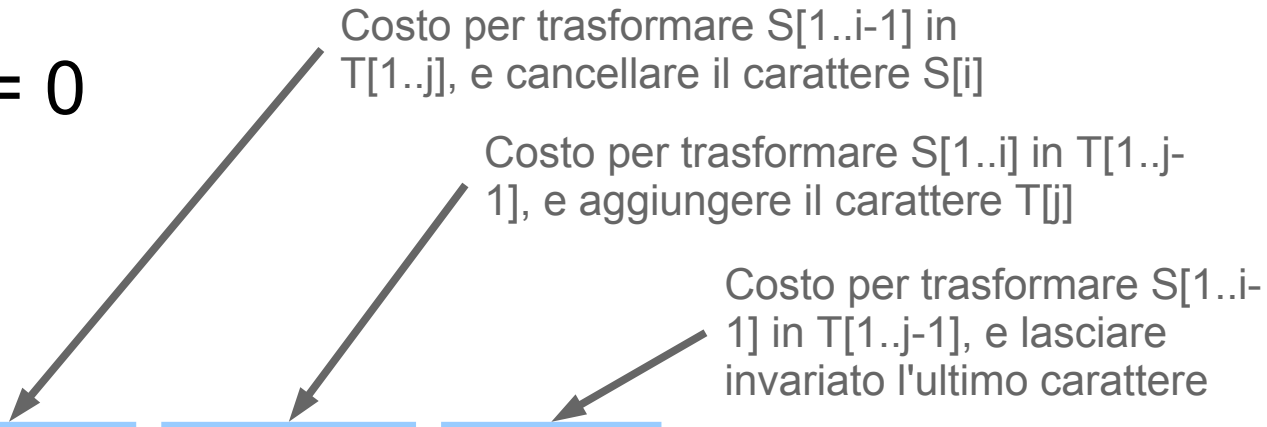






Calcolo di $L[i, j]$

- Se $i = 0$ oppure $j = 0$
  - $L[i, j] = \max \{ i, j \}$
- Altrimenti
  - Se $S[i] = T[j]$
    - $L[i, j] = \min\{ L[i-1, j] + 1, L[i, j-1] + 1, L[i-1, j-1] \}$
  - Se $S[i] \mathrel{!}= T[j]$
    - $L[i, j] = \min\{ L[i-1, j] + 1, L[i, j-1] + 1, L[i-1, j-1] + 1 \}$



Esempio

	“”	L	I	B	R	O
“”	0	1	2	3	4	5
A	1	1	2	3	4	5
L	2	1	2	3	4	5
B	3	2	2	2	3	4
E	4	3	3	3	3	4
R	5	4	4	4	3	4
O	6	5	5	5	4	3

Minimo numero di operazioni di editing necessarie per trasformare ALBE in LIB

Distanza di Levenshtein tra ALBERO e LIBRO

Conclusioni

- La programmazione dinamica è una tecnica algoritmica estremamente potente
- Va pero' applicata (dove applicabile) con disciplina
  - Identificare i sottoproblemi
  - Definire le soluzioni dei sottoproblemi
  - Calcolare le soluzioni nei casi semplici
  - Calcolare le soluzioni nel caso generale

Esercizio 1

- E' dato un insieme $X = \{1, 2, ..., n\}$ di n oggetti
  - L'oggetto i-esimo ha peso $p[i]$
  - I pesi sono numeri interi positivi
- Disponiamo di un contenitore in grado di trasportare al massimo un peso C
- Vogliamo determinare un sottoinsieme $Y \subseteq X$ tale che il peso complessivo degli oggetti in Y sia esattamente uguale a C

Esercizio 2

- E' dato un insieme $X = \{1, 2, ..., n\}$ di n oggetti
  - L'oggetto i-esimo ha peso $p[i]$
  - I pesi sono numeri interi positivi
- Disponiamo di un contenitore in grado di trasportare al massimo un peso C
- Vogliamo determinare un sottoinsieme $Y \subseteq X$ di oggetti il cui peso complessivo sia massimo possibile e minore o uguale a C (ovvero, vogliamo riempire il contenitore il più possibile)

Esercizio 3

- Disponiamo di n ≥ 1 monete aventi valori interi positivi c[1], ... c[n]; i valori delle monete sono arbitrari, quindi non necessariamente relativi ad un sistema monetario canonico.
  - Attenzione: un elemento c[i] rappresenta il valore di una singola moneta a disposizione, non infinite monete di quel valore.

  Scrivere un algoritmo basato sulla programmazione dinamica per calcolare il minimo numero di monete che è necessario usare per erogare un resto esattamente pari a R, se questo è possibile.

Esercizio 4

- Si consideri una scacchiera quadrata rappresentata da una matrice M[1..n, 1..n]. Scopo del gioco è spostare una pedina dalla casella in alto a sinistra (1, 1) alla casella in basso a destra (n, n). Ad ogni mossa la pedina può essere spostata di una posizione verso il basso, oppure di una posizione verso destra (senza uscire dai bordi).
  - Quindi, se la pedina si trova in (i, j) potrà essere spostata in (i + 1, j) oppure (i, j + 1), se possibile.

  Ogni casella M[i, j] contiene un numero reale; man mano che la pedina si muove, il giocatore accumula il punteggio segnato sulle caselle attraversate, incluse quelle di partenza e di arrivo.

Esercizio 4 - bis

- Scrivere un algoritmo efficiente che, dati in input i valori presenti nella M[1..n, 1..n] restituisce il massimo punteggio che è possibile ottenere spostando la pedina dalla posizione iniziale a quella finale con le regole di cui sopra. Ad esempio, nel caso seguente:

1	3	4	-1
3	-2	-1	5
-5	9	-3	1
4	5	2	-2

l'algoritmo deve restituire 16 (le celle evidenziate indicano il percorso da far fare alla pedina per ottenere il massimo punteggio che in questo caso è 16).